Dokumentacja wstępna TKOM

Marcel Jarosz 304006

```
Wybrany wariant ma:
- statyczne typowanie
- silne typowanie
- zmienne są domyślnie stałe
- zmienne mogą być opcjonalne
```

Opis projektu

Projekt ma na celu wykonanie interpretera własnego języka ogólnego przeznaczenia, który będzie zawierał implementację `pattern matching`

Możliwości języka

Język jest typowany silnie i statycznie. Domyślnie wszystkie zmienne są tworzone jako `immutable`. Istnieje nadanie opcjonalności danej zmiennej, przez co może ona mieć wartość `null`.

- Obsługa kilku standardowych typów danych (bool, int, double, string)
- Zmienne mogą być zdefiniowane jako opcjonalne
- Istnieje możliwość utworzenia zmiennej jako `mutable`
- Istnieje możliwość tworzenia jednolinijkowych komentarzy
- Typ znakowy `string` obsługuje escapowanie, konkatenację i multiplikację
- Obsługa operatorów matematycznych i logicznych według priorytetów, nawiasowanie
- Możliwość używania instrukcji warunkowych oraz pętli
- Możliwość definiowania własnych funkcji oraz ich wywoływanie (również rekursywne)
- Argumenty do funkcji są przekazywane przez wartość
- Tworzone zmienne mają swoje określone zakresy (np. tylko wewnątrz funkcji)
- Możliwość użycia pattern matchingu dzieki konstrukcji `match`
- Operator `??`, który umożliwia wykonanie pewnego działania zależnie czy wyrażenie jest null'em czy nie
- Obsługa operatora `as` do rzutowania wyrażeń na inne typy wbudowane
- Obsługa operatora `is` pozwalającego sprawdzić typ danego wyrażenia (czy nie ma wartości null)

Operatory

Możliwa jest budowa złożonych wyrażeń używających operatorów z predefiniowanymi priorytetami. Działania na operatorach o takich samych priorytetach będą wykonywane od lewej do prawej. Operatory według priorytetów:

1. Negacja - !
2. Operatory mnożenia

- iloczyn - a * b
- iloraz - a / b
- iloraz całkowity - a // b

- reszta z dzielenia - a % b

3. Operatory addytywne

- dodawanie - a + b
- odejmowanie - a - b

4. Operatory typów

- sprawdzenie typu - a is int
- castowanie - a as double

5. Operatory relacyjne

- porządkowe - a < b, a <= b, a > b, a >= b
- równości - a == b, a != b

6. Operator logiczny koniunkcji - a and b
7. Operator logiczny alternatywy - a or b

Słowa kluczowe

as, and, bool, break, continue, double, else, false, func, if, int, is, match, mutable, null, or, return, string, true, void, while

Gramatyka

```
program               = progStatement, { progStatement } ;

progStatement         = functionDef
                      | statement ;

statement             = conditionalStatement
                      | simpleStatement, ";"
                      | variableDeclaration, ";" ;

conditionalStatement  = ifStatement
                      | whileStatement
                      | matchStatement;

simpleStatement       = assignment
                      | functionCall
                      | returnStatement
                      | "break"
                      | "continue" ;

functionDef           = "func", identifier, "(", [parametersList], ")", ":",
functionReturnType, "{", statementBlock, "}" ;

statementBlock        = statement, {statement} ;
variableDeclaration   = ["mutable"], type, identifier, assignmentOp, (expression)
;

ifStatement           = ifBlock, [elseBlock] ;
whileStatement        = "while", "(", expression, ")", "{", {statementBlock}, "}"
;
```

```
matchStatement        = "match", "(", expression, ")", "{", insideMatchStatement,
{insideMatchStatement}. "}" ;

insideMatchStatement  = (insideMatchExpression, {(orOp | andOp),
insideMatchExpression} | "default"), "=>" simpleStatement, "," ;

insideMatchExpression = comparisonOp, valueLiteral
                      | expression
                      | "is", type;

ifBlock               = "if", "(", expression, ")", "{", {statementBlock}, "}" ;
elseBlock             = "else", "{", {statementBlock}, "}" ;

assignment            = identifier, assignmentOp, expression;

functionCall          = identifier, "(", [argumentList], ")" ;
parametersList        = parameter, {",", parameter} ;

argumentList          = expression, {",", expression} ;

parameter             = type, identifier ;
returnStatement       = "return", [expression] ;

expression            = orExpression, {"??", orExpression} ;
orExpression          = andExpression, {orOp, andExpression} ;
andExpression         = compExpression, {andOp, compExpression} ;
compExpression        = asExpression, [comparisonOp, asExpression] ;
isasExpression        = addExpression, [("is" | "as"), type] ;
addExpression         = mulExpression, {additiveOp, mulExpression} ;
mulExpression         = unaryExpression, {multiOp, unaryExpression} ;
unaryExpression       = [unaryOp], baseExpression ;
baseExpression        = valueLiteral
                      | "(", expression, ")"
                      | functionCall
                      | identifier ;

(* operators *)

unaryOp               = "!" ;
assignmentOp          = "=" ;
orOp                  = "or" ;
andOp                 = "and" ;
comparisonOp          = "=="
                      | "!="
                      | "<"
                      | "<="
                      | ">"
                      | ">=" ;
additiveOp            = "+"
                      | "-" ;
multiOp               = "*"
                      | "/"
                      | "//"
                      | "%" ;
```

```
valueLiteral          = booleanLiteral
                      | integerLiteral
                      | doubleLiteral
                      | stringLiteral
                      | "null"
                      | functionCall ;

nonNullableType       = "bool"
                      | "int"
                      | "double"
                      | "string" ;

functionReturnType    = type
                      | "void" ;

type                  = nonNullableType, ["?"] ;

booleanLiteral        = "true"
                      | "false" ;

identifier            = (letter | "_"),  letter, {digit | letter | "_"} ;

escapeLiteral         = "\\" ("t" | "b" | "r" | "n" | "\"" | "\\") ;
stringLiteral         = "\"", {charLiteral}, "\"" ;
charLiteral           = allCharacters - "\\" - "\"" | escapeLiteral ;

doubleLiteral         = integerLiteral, ".", { digit }
                      | ".", digit, {digit};

integerLiteral        = "0" | [ "-" ], naturalLiteral ;
naturalLiteral        = digitNoZero, { digit } ;

allCharacters         = ? all visible characters ? | "\t" | " " ;
letter                = "a".."z" | "A".."Z" ;

digit                 = "0"
                      | digitNoZero ;
digitNoZero           = "1" | ... | "9" ;
```

Wymagania funkcjonalne interpretera

- Możliwość wczytania, parsowania i analizy fragmentów kodu zapisanych w plikach tekstowych
- Wykonywanie wszystkich poprawnie zapisanych instrukcji
- Informowanie o wystąpieniu błędów w odpowiednich etapach analizy
- Wyświetlanie wyników działania skryptu na wyjściu terminala

Wymagania niefunkcjonalne interpretera

- Wyświetlane komunikaty o popełnionych błędach zawierają nr linii i kolumnęw którym zostały napotkane.
- Uruchamianie interpretera wyświetla informacje jeśli użytkownik popełnił błąd przy podawaniu argumentów oraz pokazuje poprawne użycie

Obsługa interpretera

Program w będzie aplikacją konsolową uruchamianą wraz z odpowiednimi parametrami (np. ścieżka do skryptu)
Wyniki działającego programu jak i napotkane błędy będą wypisywane na standardowym wyjściu.

Podział na moduły

- biblioteka pomocniczna
- lexer
- parser
- analizator semantyczny
- wykonanie

Biblioteka pomocnicza

- definicje tokenów, słów kluczowych, parametrów
- interfejsy do komunikacji pomiędzy modułami
- narzędzia pomocnicze

Analiza leksykalna

Analizator leksykalny będzie miał za zadanie rozbicie wczytanego źródła na pojedyncze tokeny. Dokona tego poprzed odczyt zawartości znak po znaku, do czasu odczytania całości sekwencji odpowiadającej poprawnemu tokenowi języka. Następnie tokeny te będą przekazywane do parsera.

Parser

Parser ma za zadanie sprawdzenie czy rozpoznane tokeny zostały ułożone według zdefiniowanej wcześniej gramatyki. Kolejne tokeny będą decydować o wybranej ścieżce wgłąb analizowanego programu. Następnie zostanie utworzone drzewo składniowe, które pozwoli na dalszą analizę skryptu.

Analizator semantyczny

Moduł ten będzie odpowiedzialny za weryfikację poprawności znaczenia wygenerowanego drzewa składniowego.
Będzie on odpowiedzialny m. in. za:

- sprawdzenie unikalności identyfikatorów
- zgodność operacji arytmetycznych i logicznych
- weryfikacja typów
- sprawdzenie sensu użycia określonych funkcji w danym kontekście

Moduł wykonawczy

Jego zadaniem jest poprawne wykonanie instrukcji zawartych w drzewie po zakończeniu poprzednich etapów analizy.

Funkcje wbudowane

- print([args]) - odpowiedzialna za wypisywanie podanych argumentów na ekran
- exit() - kończy działanie interpretera

Testowanie

Zostaną przygotowane testy akceptacyjne, do których wejściem będą fragmenty kodu napisane w omawianym języku.

Przykłady

Przykład 1

```
# function calculates Nth fibonacci number
func fib(int n) : int {
    if (n <= 1) {
        return n;
    }
    return fib(n - 2) + fib(n - 1);
}

mutable int i = 1;
int value = 13;
mutable double? sum = null;

while (i <= 10) {
    sum = (sum ?? 0.0) + fib(i) * value;
    i = i + 1;
}

string resMessage = "Sum is: " + (sum as string);
print(resMessage);
```

Przykład 2

```
func even(int value): boolean {
    return value % 2 == 0;
}

func odd_and_divisible(int value, int div): boolean {
    return value % 2 == 1 and value % div == 0;
}

string userInput = get_input() ?? "";

match(userInput as int?) {     # return value of this expression can be accessed
via '_'
```

```
    is int and even => print("Number", _, " is even"),
    is int and odd_and_divisible(_, 3) => print("Number", _,  " is odd and
divisible by 3"),
    default => print("Is not a number"),
}
```